Primer Concurso de Practica ICPC UADY [Tutorial]

Jose Santiago Vales Mena

Marzo 20, 2024

C. Cazador de Anomalías

Problema: El problema nos pide contar cuántos elementos existen en un arreglo tales que hay un elemento mayor a él.

Algoritmo bruto: La primera idea puede ser, para cada elemento, iterar sobre todo el arreglo para ver si existe alguien mayor a él, en cuyo caso lo contamos. Sin embargo, este algoritmo es $O(N^2)$ y debido a los límites de tiempo no es viable.

Solución: Basta con comparar cada número con el elemento más grande del arreglo. Entonces, podemos encontrar el mayor elemento y contar cuántos son menores.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    int N;
    cin >> N;
    vector<int> vec(N);
    int maxi = 0;

    for (auto& xi : vec) {
        cin >> xi;
        maxi = max(xi, maxi);
    }
    int ans = 0;
    for (auto& xi : vec)
        if (xi < maxi) ans++;

    cout << ans << '\n';
    return 0;
}
```

E. Escenario Oscuro

Problema: Se te da un patrón que recorrerá la matriz y debes determinar si, con los reflectores no funcionales, el patrón seguirá siendo visible por completo.

Solución: ¿Cuándo NO podremos ver el patrón completo? Cuando un elemento prendido del patrón pase por un reflector no funcional. Pero notemos que un elemento prendido del patrón pasa por TODAS las columnas del escenario. Eso quiere decir que, si existe *al menos un elemento prendido del patrón*, entonces para que funcione *todos los reflectores del escenario deben ser funcionales*; de lo contrario, no se podrá.

Por ejemplo, si el patrón es `------*-------`, el escenario debe ser `***********`, no puede tener ningún reflector disfuncional.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    int N, M, K;
    cin >> N >> M >> K;
    bool se_puede = true;
    for (int i = 0; i < N; i++) {
        string a, b;
        cin >> a >> b;

        bool reflector_disfuncional = false;
        bool elemento_patron_encendido = false;

        for(int j = 0; j < M; j++)
            if (a[j] == '-') reflector_disfuncional = true;
        for(int j = 0; j < K; j++)
            if (b[j] == '*') elemento_patron_encendido = true;

        if (reflector_disfuncional && elemento_patron_encendido)
            se_puede = false;
    }

    if (se_puede) cout << "S\n";
    else cout << "N\n";

    return 0;
}
```

B. Buscando Pokemones

Problema: Queremos encontrar el subarreglo de menor tamaño que contenga la mayor cantidad de elementos distintos.

Fuerza bruta: Primero contemos cuántos elementos distintos hay, llamemos a ese número $P_{\max}$. Queremos el subarreglo más pequeño que contenga $P_{\max}$ elementos distintos. Para cada índice i, vamos a encontrar j $(i < j)$ tal que el subarreglo desde i hasta j sea lo menor posible. Fijamos nuestro i y empezamos $i = j$. Metemos el primer elemento a un `set` y mientras no tenga la cantidad de elementos que deseo, voy recorriendo mi `j` hasta que la cantidad de elementos en mi conjunto sea $P_{\max}$.

Este enfoque tiene una complejidad de $O(N^2 \log N)$ ya que probamos todos los subarreglos en el peor de los casos.

Solución: La idea es similar a la anterior pero agregando una técnica de dos punteros. Notemos que si i_1 encontró que j_1 es su solución mínima, entonces, para $i + 1$, su respuesta debe ser *al menos* j_1. Sin embargo, al mover i a $i + 1$, estamos eliminando un elemento y no podemos simplemente hacer un `erase` de nuestro `set` ya que puede que borremos un elemento que aparece en otro lado. Por ejemplo, si tenemos la string `abcade`, nuestro set es $\{a, b, c, d, e\}$ y decidimos eliminar la primera letra del string, eso no significa borrar el elemento `a` de nuestro conjunto.

Para solucionar esto llevaremos una cubeta (podemos usar un `map` o un arreglo) donde `bucket[ai]` nos dice cuánto aparece el caracter `ai` en nuestro subarreglo. Cada vez que encontremos una aparición de elemento, aumentamos en uno nuestro bucket y cada vez que eliminemos una aparición, restamos en uno el bucket. Cuando un elemento pasa a tener frecuencia 1, la cantidad de elementos distintos que tienes aumenta en 1. Por otro lado, cuando un elemento pasa a tener frecuencia 0, la cantidad de elementos distintos disminuye en 1. Entonces, usamos esta cubeta para saber cuántos elementos distintos tenemos y usamos una técnica de dos punteros para ir variando nuestros subarreglos.

Esta solución hace que la complejidad se vuelva $O(N \log N)$.

```cpp
#include <bits/stdc++.h>
using namespace std;

unordered_map<char, int> bucket;
set<char> diff;

int N;
string S;

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    cin >> N >> S;
    for (int i = 0; i < N; i++)
        diff.insert(S[i]);

    int l = 0,
        r = 0,
        act_dif = 0,
        goal_dif = diff.size(),
        ans = N;

    while (l < N) {
        while (r < N && goal_dif != act_dif)
            if ( (++bucket[S[r++]]) == 1)
                act_dif++;
        if (act_dif == goal_dif)
            ans = min(r - l, ans);
        if ((--bucket[S[l++]]) == 0)
            act_dif--;
    }

    cout << ans << '\n';

    return 0;
}
```

D. Dando Chocolates

Problema: Tienes un arreglo de N chocolates y debes partirlos de cierta forma para que cada elemento sea entero (no dar varios pedazos a una persona) entre K personas.

Solución: Imagina que la cantidad de chocolate que quieres repartir es 2. Entonces, de la primera barra puedes obtener $\lfloor a_1/2 \rfloor$ pedazos, de la segunda $\lfloor a_2/2 \rfloor$, y en general, la cantidad de pedazos que puedes obtener de tamaño 2 de tu conjunto de chocolates es

$$\sum_{i=1}^{N} \left\lfloor \frac{a_i}{2} \right\rfloor$$

Si ese número es mayor o igual a K, entonces lo puedes repartir entre tus amigos. Entonces, podemos iterar sobre todas las posibles cantidades que podemos dar y encontrar la menor que sí podamos dar. Pero eso sería una complejidad de $O(N \cdot M)$ donde M es la máxima cantidad de chocolate que puedes dar, que en el peor de los casos se vuelve $O(N^2)$. Sin embargo, basta darnos cuenta de que si podemos dar una cantidad M_1 de chocolate, entonces podemos dar cualquier cantidad menor (no es necesario verificarlo antes). Además, si NO podemos dar una cantidad M_2 de chocolate, no vamos a poder dar algo mayor (no es necesario verificarlo). Entonces, podemos realizar una búsqueda binaria sobre el chocolate que vamos a repartir. Y eso convierte nuestra complejidad en $O(N \log M)$.

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, K;
vector<int> chocolates;

bool check(int longitud) {
    if (longitud == 0) return true;
    int suma = 0;
    for (auto& it : chocolates) {
        suma += it / longitud;
        if (suma >= K) return true;
    }
    return false;
}

int binary(int ini, int fin) {
    if (ini == fin) return ini;
    int mit = (ini + fin) / 2;
    if (!check(mit)) return binary(ini, mit);
    else return binary(mit + 1, fin);
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    cin >> N >> K;
    chocolates.resize(N);
    int maxi = 0;
    for (auto& it: chocolates) cin >> it, maxi = max(maxi, it);
    cout << binary(0, maxi) - 1;
    return 0;
}
```

A. A Más B

Problema: Sumar dos números enteros.

Solución: El problema aquí radica en que los números son MUY grandes. Usando el mejor tipo de variable de C++ que es el `unsigned long long`, llegamos a los más a 10^{19}. Notemos que los límites de este problema son 10^{10^5}. Entonces, ni siquiera podemos leer el propio número. Lo que debemos hacer es leer el número como una `string` e implementar nuestra función suma. ¿Recuerdas cómo sumar a mano? Básicamente es lo que debemos implementar. Otra opción es utilizar Python, que tiene suma de números grandes integrada.

```cpp
#include <bits/stdc++.h>
using namespace std;

struct BigInteger {
    string str;

    BigInteger(string s) { str = s; }

    BigInteger operator+(const BigInteger& b)
    {
        string a = str;
        string c = b.str;
        int alen = a.length(), clen = c.length();
        int n = max(alen, clen);
        if (alen > clen)
```

```
            c.insert(0, alen - clen, '0');
        else if (alen < clen)
            a.insert(0, clen - alen, '0');
        string res(n + 1, '0');
        int carry = 0;
        for (int i = n - 1; i >= 0; i--) {
            int digit = (a[i] - '0') + (c[i] - '0') + carry;
            carry = digit / 10;
            res[i + 1] = digit % 10 + '0';
        }
        if (carry == 1) {
            res[0] = '1';
            return BigInteger(res);
        }
        else {
            return BigInteger(res.substr(1));
        }
    }
};

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    string a, b; cin >> a >> b;

    BigInteger a0(a);
    BigInteger b0(b);
    BigInteger sum = a0 + b0;

    cout << sum.str << '\n';
    return 0;
}
```

F. Frase Flotante

Problemas: Encontrar cuántas rotaciones a la izquierda debes hacer para llegar de una string a otra.

Solución: Si concatenamos una string consigo misma, resulta que todas las substrings de tamaño N son rotaciones de la string original. Por ejemplo, digamos que tenemos la string `abcdef` y la concatenamos consigo misma `abcdefabcdef`. Ahora, si tomamos cualquier subcadena de tamaño N, es justamente una rotación a la izquierda. Ahora simplemente debemos encontrar dónde aparece la segunda cadena en ese texto. Durante el concurso, los casos no eran muy fuertes y la solución de fuerza bruta de verificar cada cadena una por una daba AC, pero esa solución es cuadrática $O(N^2)$. Sin embargo, podemos usar el algoritmo de KMP para encontrarlo en $O(N)$, que era la solución esperada.

```
#include <bits/stdc++.h>
using namespace std;

vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
```

```
    return pi;
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    string a, b;
    cin >> a >> b;
    string c = b + "#" + a + a;
    int N = a.size();
    auto kmp = prefix_function(c);
    for (int i = 0; i < N; i++) {
        if (kmp[2 * N + i] == N) {
            cout << i;
            return 0;
        }
    }

    return 0;
}
```